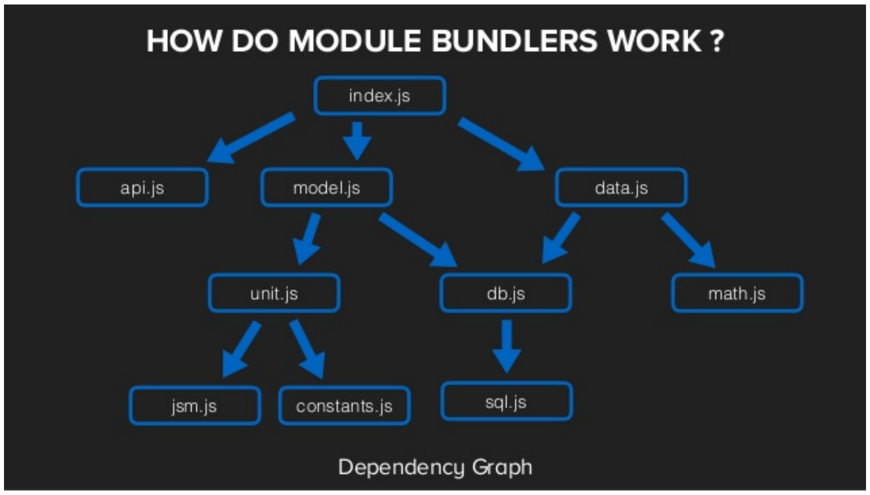
Date=2/11/2020
Lecture By=Manish Mahant
Subject ⇒ Bundlers And Task Runners

JavaScript Module Bundlers

Module bundlers are tools that process your modern JavaScript applications, internally build dependency graphs which map every modules your projects need and generate one or module bundles.

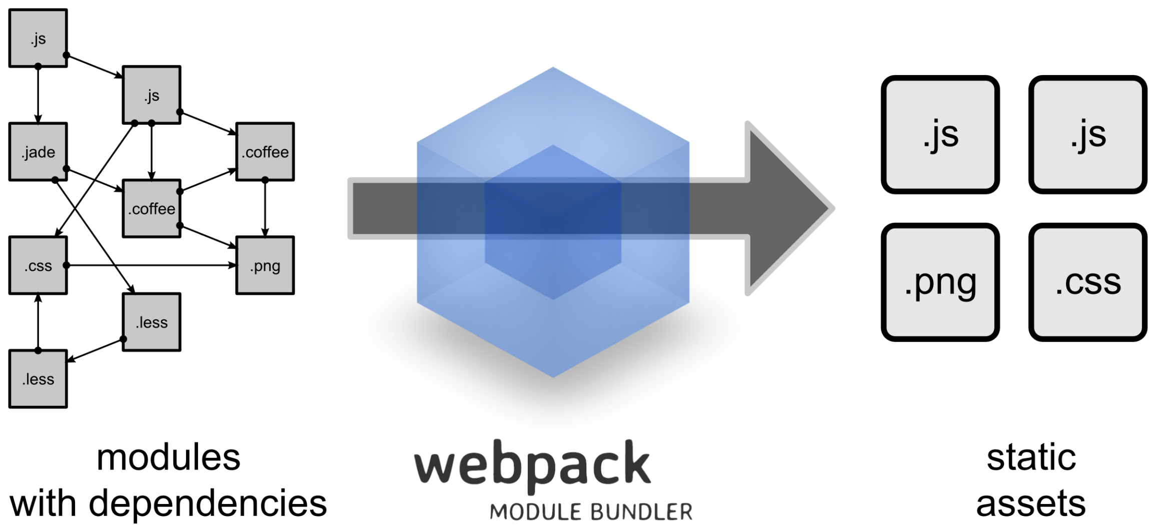
They're typically different in terms of how many kind of non-JavaScript files (css, json, png, jpeg, xml, etc.) they can process, whether they'll output npm packages or complete applications, and how much custom configurations they support.

They're not general-purpose task runners or built but can be used like those with very limited supported behavior (like webpack focusing on automating frontend tasks and generate a website). They often have but not limited following features:



```
- Bundle JavaScript code in different module formats
- Bundle HTML, CSS, images, and other file assets
- Target outputs can be packages or complete complications
- Transformed code and assets can be compatible to one or more environments
- Build optimizations like tree-shaking, code-splitting, scope-hoisting, etc.
- Hot module replacement during development
- Custom configuration for advanced use cases
- Hashed output filenames for caching strategy
- Generating source maps and developer friendly logging
```

What Is Webpack

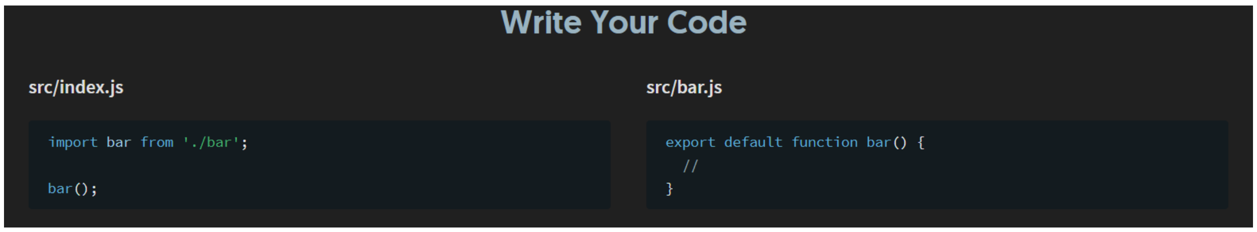


Webpack is the most popular and powerful module bundler for JavaScript when they can process any kind of files and they can output packages, websites, or servers.

They have been well developed with flexible architecture including highly customizable configurations, extensive plugins and loaders ecosystem.

Out of the box, webpack only understands JavaScript and JSON files. *Loaders* allow webpack to process other types of files and convert them into valid modules that can be consumed by your application and added to the dependency graph.

While loaders are used to transform certain types of modules, plugins can be leveraged to perform a wider range of tasks like bundle optimization, asset management and injection of environment variables.



Write Your Code

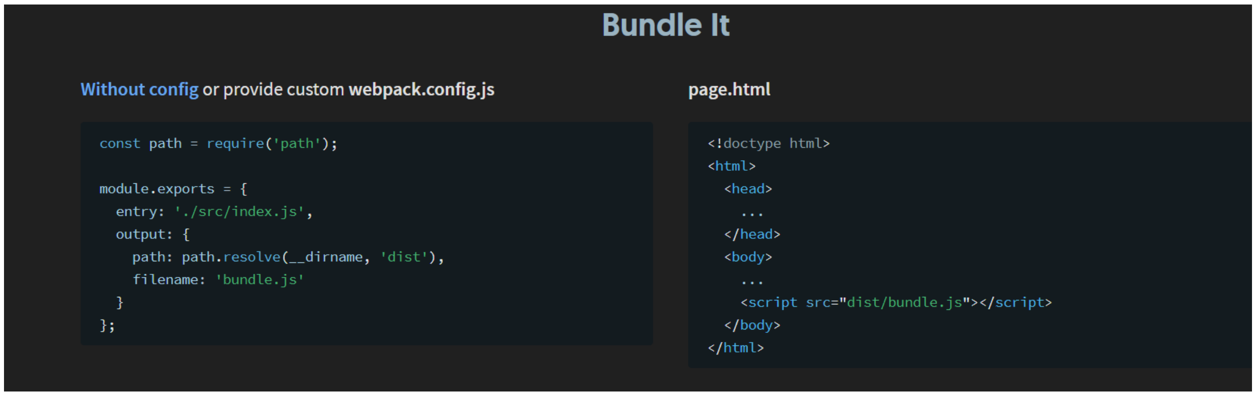
src/index.js

```
import bar from './bar';

bar();
```

src/bar.js

```
export default function bar() {
  //
}
```


Bundle It

Without config or provide custom webpack.config.js

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js'
  }
};
```

page.html

```
<!doctype html>
<html>
  <head>
    ...
  </head>
  <body>
    ...
    <script src="dist/bundle.js"></script>
  </body>
</html>
```

Javascript Transpilers

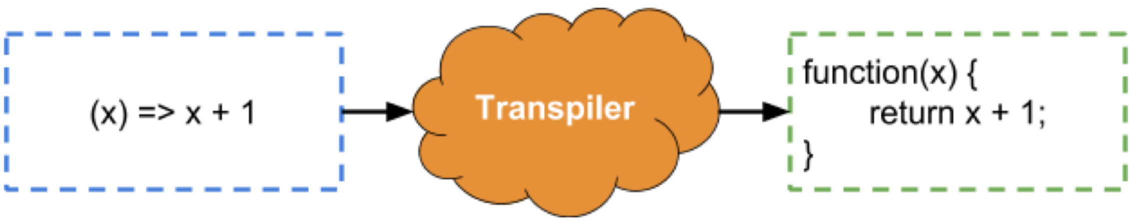
Transpilers, or **source-to-source** compilers, are tools that read source code written in one programming language, and produce the equivalent code in another language. Languages you write that transpile to JavaScript are often called **compile-to-JS** languages, and are said to **target** JavaScript.

Babel Transpiler

Babel is a JavaScript **transpiler** that converts edge JavaScript into plain old ES5 JavaScript that can run in any browser (even the old ones). It makes available all the syntactical sugar that was added to JavaScript with the new ES6 specification, including classes, fat arrows and multiline strings.
How does Babel transpiler work?

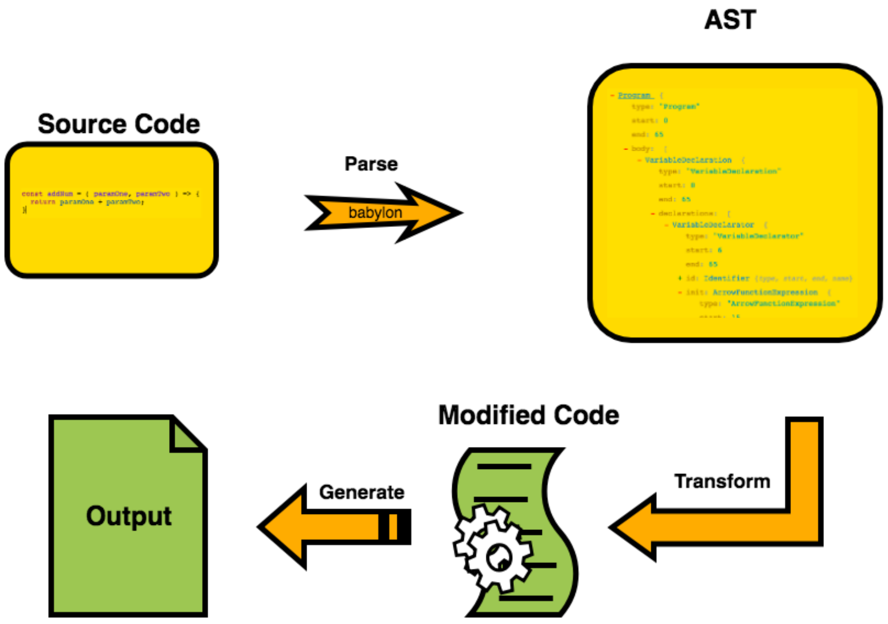
How Babel Transpiler Works

Babel-transpiler converts the syntax of modern JavaScript into a form, which can be easily understood by older browsers. For example, arrow function, const, let classes will be converted to function, var, etc.



```
(x) => x + 1
```

```
function(x) {
        return x + 1;
}
```

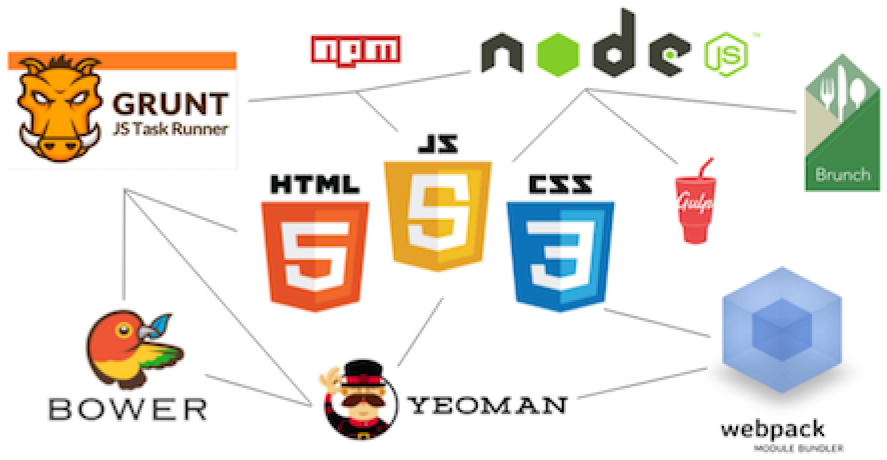


```
const addNum = ( paramOne, paramTwo ) => {
  return paramOne + paramTwo;
}
```

```
- Program {
    type: "Program"
    start: 0
    end: 65
  - body: [
    - VariableDeclaration {
        type: "VariableDeclaration"
        start: 0
        end: 65
      - declarations: [
        - VariableDeclarator {
            type: "VariableDeclarator"
            start: 6
            end: 65
          + id: Identifier {type, start, end, name}
          - init: ArrowFunctionExpression {
              type: "ArrowFunctionExpression"
```

Task Runners In Javascript

A **JS task runner** basically runs commands for you that would otherwise be tedious or impossible. They do things like compile your code from SCSS to CSS or TypeScript to **JavaScript**. ... **Task runners** are the most powerful innovation to arrive in the **JS** ecosystem in a while.



Gulp Task Runner

Introduction

Modern web development has many repetitive tasks like running a local server, minifying code, optimizing images, preprocessing CSS and more. This text discusses gulp, a build tool for automating these tasks.

What is gulp?

Gulp is a cross-platform, streaming task runner that lets developers automate many development tasks. At a high level, gulp reads files as streams and pipes the streams to different tasks. These tasks are code-based and use plugins. The tasks modify the files, building source files into production files. To get an idea of what gulp can do check the list of gulp recipes on GitHub.

How to set up gulp

Setting up gulp for the first time requires a few steps.

Node

Gulp requires Node, and its package manager, npm, which installs the gulp plugins.

If you don't already have Node and npm installed, you can install them with Node Version Manager (nvm). This tool lets developers install multiple versions of Node, and easily switch between them.

Nvm can then be used to install Node by running the following in the command line:

```
nvm install node
```

This also installs Node's package manager, npm. You can check that these are both installed by running the following commands from the command line:

```
node -v
```

```
npm -v
```

If both commands return a version number, then the installations were successful.

Gulp command line tool

Gulp's command line tool should also be installed globally so that gulp can be executed from the command line. Do this by running the following from the command line:

```
npm install --global gulp-cli
```

Creating a new project

Before installing gulp plugins, your application needs to be initialized. Do this by running the following command line command from within your project's working directory:

```
npm init
```

This command begins the generation of a **package.json** file, prompting you with questions about your application. For simplicity these can all be left blank (either by skipping the prompts with the return key or by using `npm init -y` instead of the above command), but in production you could store application metadata here. The file looks like this (your values may be different):

package.json

```
{
  "name": "test",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Don't worry if you don't understand what all of these values represent, they are not critical to learning gulp.

This file is used to track your project's packages. Tracking packages like this allows for quick reinstallation of all the packages and their dependencies in future builds (the `npm install` command will read **package.json** and automatically install everything listed).

Installing packages

Gulp and Node rely on plugins (packages) for the majority of their functionality. Node plugins can be installed with the following command line command:

```
npm install pluginName --save-dev
```

This command uses the npm tool to install the `pluginName` plugin. Plugins and their dependencies are installed in a **node_modules** directory inside the project's working directory.

The `--save-dev` flag updates **package.json** with the new package.

The first plugin that you want to install is gulp itself. Do this by running the following command from the command line from within your project's working directory:

```
npm install gulp --save-dev
```

Gulp and its dependencies are then present in the the **node_modules** directory (inside the project directory). The **package.json** file is also updated to the following (your values may vary):

package.json

```
{
  "name": "test",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "gulp": "^3.9.1"
  }
}
```

Note that there is now a `devDependencies` field with gulp and its current version listed.

Gulpfile

Once packages are installed (in **node_modules**), you are ready to use them. All gulp code is written in a **gulpfile.js** file. To use a package, start by including it in **gulpfile.js**. The following code in your gulpfile includes the gulp package that was installed in the previous section:

gulpfile.js

```
var gulp = require('gulp');
```

Creating tasks

Gulp tasks are defined in the **gulpfile.js** file using `gulp.task`. A simple task looks like this:

gulpfile.js

```
gulp.task('hello', function() {
  console.log('Hello, World!');
});
```

This code defines a `hello` task that can be executed by running the following from the command line:

```
gulp hello
```

A common pattern for gulp tasks is the following:

1. Read some source files using `gulp.src`
2. Process these files with one or more functions using Node's `pipe` functionality
3. Write the modified files to a destination directory (creating the directory if doesn't exist) with `gulp.dest`
4. `gulp.task('task-name', function() {`
5. `  gulp.src('source-files') // 1`
6. `  .pipe(gulpPluginFunction()) // 2`

```
  .pipe(gulp.dest('destination')); // 3
});
```

A complete gulpfile might look like this:

gulpfile.js

```
// Include plugins
var gulp = require('gulp'); // Required
var pluginA = require('pluginA');
var pluginB = require('pluginB');
var pluginC = require('pluginC');

// Define tasks
gulp.task('task-A', function() {
  gulp.src('some-source-files')
  .pipe(pluginA())
  .pipe(gulp.dest('some-destination'));
});

gulp.task('task-BC', function() {
  gulp.src('other-source-files')
  .pipe(pluginB())
  .pipe(pluginC())
  .pipe(gulp.dest('some-other-destination'));
});
```

Where each installed plugin is included with `require()` and tasks are then defined using functions from the installed plugins. Note that functionality from multiple plugins can exist in a single task.

Questions For Self Practice / CC/Assignment For The Day

Coding-challenge
https://au-assignment.s3.ap-south-1.amazonaws.com/Week_19_Day_1_Challenge-4658edb5-1c3c-4e9b-b407-992b3ca6e9b0.pdf

Assignment
https://au-assignment.s3.ap-south-1.amazonaws.com/Week_19_Day_1_Assignment-ca13b704-d1d5-493f-b754-0214bf294592.pdf